5 Discrete Logarithms and Key Exchange

5.1 Key Exchange

One of the major limitations on the algorithms we've discussed—and many we haven't—is the problem of *key exchange*. Any symmetric encryption algorithm requires the two parties to exchange a key in secret. This can often be difficult, since the whole point of cryptography is to protect your communications from evesdroppers.

Modern algorithms allow us to exchange keys in public, in a way that does not give evesdroppers access to the key even if they can overhear and understand the message. An algorithm that enables this is called a key exchange algorithm.

We illustrate the basic idea with a physical analogue. Suppose you want to send presents to a friend, but need to send them inside a locked box—otherwise the presents will be stolen out of the box during transit. This is a problem, since you don't have any locks that your friend has a key to. So you need to find a way to send the friend a key—without the key itself being stolen out of the locked box.

One solution is to pack a key inside a box that can be locked with two separate padlocks. Lock one, and then send the box to your friend. Your friend can't open the box, but they can put their own padlock on the box. Now neither of you can open it alone.

Your friend sends the box back to you, and you remove your lock. Send the box back to your friend, and they can remove their own lock and now retrieve the key you originally packed. You have successfully exchanged a key without ever sending it unlocked, despite the fact that you and your friend originally did not have a shared key.

Our task for the day will be to find a mathematical way to implement this idea.

5.2 Merkle's Puzzles

The first variant of this idea was developed by Ralph Merkle in 1974. He suggested the following algorithm:

Algorithm 5.1.

1. Bob generates N different symmetric keys and attaches an identification code i to each of them.

2. For each key, Bob encrypts a message of the form "This is the ith key on my list. The key is K_i." He uses an encryption algorithm that is possible but computationally expensive to brute force.

3. Bob sends Alice all N of the messages generated this way. Alice chooses one at random and brute-force decrypts it, and sends the identifier to Bob.

4. Bob and Alice can now communicate using the symmetric key they have agreed on.

Notice that Alice only needs to brute-force decrypt *one* of the messages Bob sends. However, if Eve does not know which message Alice chose, so if she intercepts Bob's transmission, she must decrypt all of them to find the identifiers and know the¡ key Alice and Bob are using.

However, while Eve's job here is harder than Alice's, it's not enough harder—it's at most quadratically harder, which isn't generally considered a good enough speed advantage for secure cryptography.

5.3 The Discrete Logarithm Problem

Recall that in the real numbers, the logarithm $\log_a(b)$ is the (unique) solution to the equation $a^x = b$. We can define an analog of this operation in modular arithmetic, after we lay some groundwork.

Definition 5.1. Let m be a positive integer. We call a number $a \in \mathbb{Z}/m\mathbb{Z}$ a *unit* modulo m if a has an inverse modulo m. We denote the set of units modulo m by $\mathbb{Z}/m\mathbb{Z}^\times$.

If p is a prime, then $\mathbb{Z}/p\mathbb{Z}^\times = \{1, 2, \dots, p-1\}$.

By Fermat's Little Theorem, we know that if $g \in \mathbb{Z}/p\mathbb{Z}^\times$ then $g^{p-1} \equiv 1 \mod p$.

Definition 5.2. We say that a number $g \in \mathbb{Z}/p\mathbb{Z}$ is a *primitive root* modulo p if the set $\{g, g^2, g^3, \dots, g^{p-1}\} = \mathbb{Z}/p\mathbb{Z}^\times$—that is, if raising g to successive powers gives every unit modulo p.

Necessarily, if the $p-1$ elements of $\{g, g^2, \dots, g^{p-1}\}$ hit all $p-1$ elements in $\mathbb{Z}/p\mathbb{Z}$, then there are no repetitions.

Example 5.3. 3 is a primitive root mod 7, since we have $3, 9 \equiv 2, 6, 18 \equiv 4, 12 \equiv 5, 15 \equiv 1$.

2 is not a primitive root mod 7, since we get $2, 4, 8 \equiv 1$ and we hit a repetition before we get every element of $\mathbb{Z}/7\mathbb{Z}^\times$.

But 2 is a primitive root mod 11, since we get $2, 4, 8, 16 \equiv 5, 10, 20 \equiv 9, 18 \equiv 7, 14 \equiv 3, 6, 12 \equiv 1$.

There's no algorithm for finding primitive roots; all you can do is keep trying until one of them works. 2 often but not always works. We can reduce the time it takes to test a number with the following fact:

Fact 5.4. *If $g \in \mathbb{Z}/p\mathbb{Z}^\times$ then $\#\{g^i \mod p : 1 \leq i \leq p-1\} | p-1$.*

Thus in particular, if we compute $g, g^2, \dots, g^{(p-1)/2}$ and we haven't found a number equivalent to 1, then we know g is a primitive root.

Definition 5.5. Let p be prime, g a primitive root mod p, and $h \in \mathbb{Z}/p\mathbb{Z}^\times$. Then if x is a natural number such that $g^x \equiv h \mod m$, we say that x is a *discrete logarithm of h to the base g modulo m.* The problem of finding a solution to this congruence is the *Discrete Logarithm Problem.*

Some authors will call this the *index* of h with respect to g, denoted $\operatorname{ind}_g(h)$.

As posed here, the Discrete Logarithm Problem always has a solution. In fact it will always have infinitely many, because if x is a solution then $x + n(p-1)$ is also a solution for any $n \in \mathbb{Z}$. We generally take "the" solution to have the property that $0 \leq x \leq p-2$. (Notice that this same problem comes up in complex analysis, where the logarithm is only defined modulo $2\pi i$).

Example 5.6. We know that 2 is a primitive root mod 11. So what is $\log_2(6)$ mod 11?

We compute $2, 4, 8, 16 \equiv 5, 10, 20 \equiv 9, 18 \equiv 7, 14 \equiv 3, 6$, so $\log_2(9) = 4$.

Example 5.7. 3 is a primitive root mod 7. What is $\log_3(4)$ mod 7?

We compute $3, 9 \equiv 2, 6, 18 \equiv 4$ so $\log_3(4) = 4$.

Example 5.8. 2 is a primitive root mod 29. What's $\log_2(7)$ mod 29?

We compute $2, 4, 8, 16, 32 \equiv 3, 6, 12, 24, 48 \equiv 19, 38 \equiv 9, 18, 36 \equiv 7$. So $\log_2(7) = 12$.

Remark 5.9. Like the problem of finding a primitive root, there isn't really a much better way to find the discrete logarithm of a number than just trying things until one works. You can get a speed up from $O(p)$ (trying all $p-2$ possibilities), to to $O(\sqrt{p})$ with clever algorithms, but that doens't help as much as you'd think against the inherent exponential growth.

The modular logarithm maintains many of the same properties that the real-number logarithm has.

Fact 5.10. *1.* $\log_g(1) = 0$

2. $\log_g(ab) \equiv \log_g(a) + \log_g(b) \mod p - 1$

3. $\log_g(a^r) \equiv r \log_g(a) \mod p - 1$.

Remark 5.11. Property (2) tells us that $\log_g$ is a group isomorphism from $\mathbb{Z}/p\mathbb{Z}^\times$ to $\mathbb{Z}/(p-1)\mathbb{Z}$.

5.4 Diffie-Hellman key exchange

Diffie-Hellman key exchange is an algorithm first published by Diffie and Hellman in 1976. (It was actually discovered in 1975 by James H. Ellis, Clifford Cocks, and Malcolm J. Williamson of British intelligence, but their discovery wasn't declassified until 1997).

Algorithm 5.2. Alice and Bob wish to exchange a key. They follow the following steps:

1. Choose a large prime p, and a non-zero integer $g \in \mathbb{Z}/p\mathbb{Z}^\times$.

2. Alice chooses a secret integer a, and Bob chooses a secret integer b. Neither party reveals this integer to anyone.

3. Alice computes $A \equiv g^a \mod p$ and Bob computes $B \equiv g^b \mod p$, and they (publicly) exchange these values with each other.

4. Now Alice computes $A' \equiv B^a \mod p$ and Bob computes $B' \equiv A^b \mod p$.

5. $A' \equiv B' \mod p$, so Alice and Bob use this shared information as their key.

Remark 5.12. Technically this doesn't exchange a key so much as create a commonly known key. But it does allow Alice and Bob to mutually share information to get a mutual key, without sharing the key with evesdroppers.

Proposition 5.13. *$A' \equiv B' \mod p$.*

Proof. $A' \equiv B^a \equiv (g^b)^a \equiv (g^a)^b \equiv A^b \equiv B' \mod p$. □

Example 5.14. A toy example: Suppose Alice and Bob have chosen the prime 29 and the primitive root 2. (These are terrible choices for security). Alice chooses a secret key $a = 7$ and Bob chooses a secret key $b = 17$. Then Alice computes $A = g^a = 2^7 \equiv 12 \mod 29$; and Bob computes $B = g^b = 2^{17} \equiv 21 \mod 29$.

Alice sends $A = 12$ to Bob, and Bob sends $B = 21$ to Alice. Eve can observe both these numbers, which are public knowledge.

Then Alice computes $A' = B^a = 21^7 \equiv 12 \mod 29$, and Bob computes $B' = A^b = 12^{17} \equiv 12 \mod 29$. So Alice and Bob have a shared secret of 12.

Example 5.15. Suppose Alice and Bob are working with the prime modulus $p = 941$ and primitive root $g = 627$. Alice chooses a secret key $a = 347$, computes $A = g^a = 627^{347} \equiv 390$ mod 941; and Bob chooses $b = 781$ and compute $B = g^b = 627^{781} \equiv 691 \mod 941$.

Alice sends the number 390 to Bob, and Bob sends the number 691 to Alice. Both of these numbers are visible to Eve and are public knowledge. Then Alice computes

$$A' = B^a = 691^{347} \equiv 470 \mod 941.$$

And Bob computes

$$B' = A^b = 390^{781} \equiv 470 \mod 941.$$

Now Alice and Bob have a secret number 470 in common that Eve doesn't know.

So how secure is this, really? Remember that for any sort of encryption algorithm, we want it to be easy for Alice and Bob to compute, but expensive for Eve to compute. We first need to talk about what it means for a computation to be easy or hard.

5.5 Complexity and big-O notation

Definition 5.16. Let $f(x)$ and $g(x)$ be positive functions of x. We say that f is big-O of g, and write $f(x) = O(g(x))$, if there are positive constants c, C such that $f(x) \leq cg(x)$ for all $x \geq C$.

This represents the idea that when x is very big, $f(x)$ will not be much bigger than $g(x)$.. Generally we'll use this when f is a relatively complicated function, and we can replace it with a simpler function g.

Proposition 5.17. *If* $\lim_{x\to\infty} \frac{f(x)}{g(x)}$ *exists and is finite, then* $f(x) = O(g(x))$.

Example 5.18. If $f(x)$ is bounded for all $x \geq C$, then we write $f(x) = O(1)$.

It's easy to see that $x = O(x^2)$. It's also the case that $7x^2 + 5x + 3 = O(x^2)$.

$x^n = O(2^x)$ for any natural number n.

When we study algorithms, we use big-O notation to describe the running time of algorithms. In particular, we want to compare the number of steps an algorithm takes to the number of *bits* in the input, since computers operate in bits. (Importantly, this means that if our algorithm takes a number n as input, we don't want to think about the size of the number n; instead we want to think about the number of bits it takes to represent n, which is approximately $\log_2(n)$).

If an algorithm takes $f(k)$ steps when given k bits as input, and $f(kn) = O(g(k))$, then we say that the algorithm has a running time of $O(g(k))$.

If an algorithm is $O(k^\ell)$ for some constant $\ell > 0$, we say that the algorithm runs in *polynomial time*. (If $\ell = 1$ then we say the algorithm runs in *linear time*; if $\ell = 2$ we say it runs in *quadratic time*, and so on). We consider polynomial-time algorithms to be "fast".

If an algorithm is $O(2^{ck})$ for some $c > 0$, we say the algorithm runs in *exponential time*. Exponential time algorithms are considered "slow". Thus in the ideal cryptosystem, encryption and decryption knowing the key will be polynomial-time computations, but decryption without the key will be an exponential-time computation.

There is a third important category of *subexponential time* algorithms. These algorithms are $O(2^{\epsilon k})$ for every $\epsilon > 0$. Thus they're faster than exponential, but not necessarily as fast as polynomial algorithms. For instance, an algorithm that runs in $O(2^{\sqrt{k}})$ time is subexponential, but significantly slower than polynomial.

Remark 5.19. I should make a remark here about the famous NP complexity class. A problem is said to be "in NP" if it's possible to verify a solution in polynomial time, but not necessarily possible to find one in polynomial time. It's currently believed but not known that $P \neq NP$—that is, there are problems whose answers can be verified in polynomial time, but not found in polynomial time.

Almost all reasonable cryptography is in NP. Since decryption with a known key should be doable in polynomial time, we can always check if a key is correct by trying it out. So most cryptography relies on the belief that $P \neq NP$.

5.6 Security of Diffie-Hellman

First let's look at the computation Alice and Bob need to do. Each of them needs to do exponentiations: Alice first computes g^a and then she computes B^a. (Bob computes g^b and A^b, which is exactly the same process, so we'll just look at Alice for simplicity).

So how do we do a large exponentiation mod p? The obvious and naive method is to compute $g, g^2, g^3, g^4, \dots, g^a$. So in our toy example 5.14 Alice would compute

$$\begin{array}{llll} 2^1 = 2 & 2^2 = 4 & 2^3 = 8 & 2^4 = 16 \\ 2^5 = 32 \equiv 3 & 2^6 \equiv 6 & 2^7 \equiv 12. & \end{array}$$

This is totally manageable when a and p are this small, but when they get bigger, this leads to a large number of multiplications. Alice will have to conduct a multiplications to compute

g^a, and another a multiplications to compute B^a, so in total she will need to conduct $2a$ multiplications. This algorithm is thus $O(a)$.

But this is actually an exponential algorithm! Remember we don't care about the size of the number that is the input; we care about the number of bits. And the number of bits involved is $k \approx \log_2(a)$. Since $a = 2^k$ by definition, we see that this algorithm is $O(2^k)$, and thus exponential time.

Fortunately, Alice has an easier algorithm.

Algorithm 5.3.

1. Compute g^{2^k} for $2^k \leq a$. That is, compute $g, g^2, g^4, g^8, \dots, g^{2^k}$. We can do this by repeated squaring, without computing intermediate powers.

2. Now express the exponent a in binary. That is, write $a = c_0 + c_1 \cdot 2 + c_2 \cdot 2^2 + \dots + c_k 2^k$, where $c_i \in \{0, 1\}$.

3. Now we can compute

$$\begin{aligned} g^a &= g^{c_0 + c_1 \cdot 2 + c_2 \cdot 2^2 + \dots + c_k 2^k} = g^{c_0} g^{c_1 \cdot 2} g^{c_2 \cdot 2^2} \cdots g^{c_k 2^k} \\ &= g^{c_0} (g^2)^{c_1} (g^{2^2})^{c_2} \cdots (g^{2^k})^{c_k}. \end{aligned}$$

   But we already know g^{2^i} for each i, and the c_i are all either 0 or 1 so don't involve any computation. So we only have to multiply up to k things together here.

With this algorithm, we do k squarings to compute the g^{2^i}, and we do k multiplications to finally compute g^a, so we need to do $2k$ total multiplications and this algorithm is $O(k)$. Since k is the number of bits of input, this algorithm is a linear-time algorithm.

Example 5.20. Let's see how this applies to our toy example 5.14. Alice wanted to compute $2^7 \mod 29$.

We first compute every 2^kth power of g, so we compute

$$2^1 = 2 \qquad 2^2 = 4 \qquad 2^4 = 16 \qquad 2^8 = 256 \equiv 24.$$

Then we write a in binary; that is, we write $7 = 1 + 2 + 2^2$. So we have

$$2^7 \equiv 2^{2^0} \cdot 2^{2^1} \cdot 2^{2^2} \equiv 2 \cdot 4 \cdot 16 \equiv 64 \equiv 12.$$

So Alice and Bob can do their work naively in about $O(2^k)$ time, or cleverly in $O(k)$. What about Eve?

Remember that Eve sees the numbers $A \equiv g^a \mod p$ and $B \equiv g^b \mod p$. She needs a way to find $A' = B' \equiv g^{ab} \mod p$. How can she do this?

The only *known* way to solve this problem is to compute at least one of a or b. We haven't proven that there's not a more efficient algorithm, but we don't know of one. So in practice, Eve wants to solve the following problem:

Discrete Logarithm Problem: Given a modulus p and a primitive root g, and a number A, find an integer x such that $g^x \equiv A \mod p$.

Once Eve has solved this problem, she has all the information that Alice does.

There is again a naive algorithm for this. Compute $g, g^2, g^3, \dots \mod p$, and stop when you get A as an output. This algorithm will take about a multiplications. Thus the algorithm is $O(a) = O(2^k)$, and exponential.

We do in fact know a better algorithm for solving the Discrete Logarithm problem. Fortunately for cryptography purposes, it's not *much* better.

Algorithm 5.4 (Shanks's Babystep-Gianstep Algorithm)**.** Suppose we have a prime number p and a primitive root g, and an integer A, and we want to find an integer x such that $g^x \equiv A \mod p$. Then

1. let $n = 1 + \lfloor \sqrt{p} \rfloor$. Thus $n > \sqrt{p}$.
2. (Baby steps) Calculate $g^0, g^1, g^2, \dots, g^n \mod p$. Find an inverse for $g^n \mod p$.
3. (Giant steps) Calculate $A, A \cdot g^{-n}, A \cdot g^{-2n}, \dots, A \cdot g^{-n^2} \mod p$.
4. Find a match between these two lists, so that we have $g^i \equiv hg^{-jn}$.
5. Then $x = i + jn$ is a solution to $g^x \equiv h \mod p$.

Remark 5.21. This algorithm is named after Daniel Shanks, and was first discovered by Alexander Gelfond in 1962.

Proposition 5.22. *Shanks's algorithm solves the discrete logarithm problem in* $O(\sqrt{p} \cdot \log_2 p)$ *time.*

Proof. First we show that the algorithm is $O(\sqrt{p} \cdot \log_2 p)$. The lists in steps 2 and 3 each involve about n computations, so we have $2n \approx 2\sqrt{p}$ computations there. Each computation will less than $\log_2 p$ steps.

Step 4 requires finding matches between two lists; this is a standard computer science task, and takes about $\log_2$ the length of the lists, and thus is $O(\log_2 p)$. This winds up trivial when added to the previous steps. So the whole algorithm is $O(\sqrt{p} \cdot \log_2 p)$.

Now let's prove the algorithm works. If x is the solution to $g^x \equiv A \mod p$, we can write $x = nq + r$ for $0 \leq r < n$ by the division algorithm. Since $a \leq x < p$, we know that $q = \frac{x-r}{n} < \frac{N}{n} < n$ since $n > \sqrt{N}$.

So $g^x \equiv A \mod p$ is the same as $g^r \equiv A \cdot g^{-qn} \mod p$, restricted to $0 \leq r < n$ and $0 \leq q < n$. Our list from step 2 is a list of g^r, and our list from step 3 is a list of $A \cdot g^{-qn}$. So these lists will have a common element, and finding it will give us $x = r + qn$. □

Example 5.23. 23 is prime, and 10 is a primitive root modulo 13. Let's solve $10^x \equiv 7 \mod 23$.

Following Shanks's algorithm, we see that $4 < \sqrt{23} < 5$ so we set $n = 5$.

We compute $1, 10, 10^2, 10^3, 10^4, 10^5$:

10^0	10^1	10^2	10^3	10^4	10^5
1	10	8	11	18	19

We need an inverse of $10^5 \equiv 19 \equiv -4$; we see that -6 will be an inverse modulo 23, so our inverse is -6 or 17.

Now we compute $7 \cdot 2^i$:

$7 \cdot (-6)^0$	$7 \cdot (-6)^1$	$7 \cdot (-6)^2$	$7 \cdot (-6)^3$	$7 \cdot (-6)^4$	$7 \cdot (-6)^5$
7	4	22	6	10	9

We find a 10 on both lists, corresponding to $r = 1$ and $q = 4$. Thus we have $x = qn + r = 4 \cdot 5 + 1 = 21$. And we check that indeed, $10^{21} \equiv 7 \mod 23$.

Example 5.24. 37 is a prime number, and 5 is a primitive root modulo 73. Let's solve $5^x \equiv 13 \mod 37$.

Following Shanks's algorithm, we see that $6 < \sqrt{37} < 7$ so we set $n = 7$.

We compute $1, 5, 5^2, \ldots, 5^7$: we get

5^0	5^1	5^2	5^3	5^4	5^5	5^6	5^7
1	5	25	14	33	17	11	18

We find an inverse of $5^7 \equiv 18$. We see that $2 \cdot 18 = 36 \equiv -1$, so the inverse of 18 is -2.

Now we compute $13 \cdot 5^{-in}$ and we get

13	$13 \cdot 5^{-7}$	$13 \cdot 5^{-14}$	$13 \cdot 5^{-21}$	$13 \cdot 5^{-28}$	$13 \cdot 5^{-35}$	$13 \cdot 5^{-42}$	$13 \cdot 5^{-49}$
13	$13 \cdot -2$	$13 \cdot (-2)^2$	$13 \cdot (-2)^3$	$13 \cdot (-2)^4$	$13 \cdot (-2)^5$	$13 \cdot (-2)^6$	$13 \cdot (-2)^7$
13	11	15	7	23	28	18	1

We see that there's a matching 11 on both lists, corresponding to $r = 6$ and $q = 1$, so we have $x = 1 \cdot 7 + 6 = 13$. And we can check that $5^{13} \equiv 13 \mod 37$, as desired.

References

[Singh(2015)] Vikram Singh. A practical key exchange for the internet using lattice cryptography. Cryptology ePrint Archive, Report 2015/138, 2015. http://eprint.iacr.org/2015/138.